**Task 1 - Das Experiment**

**Cost**: 1 points.

Create HashMap<Integer, Integer>. The first thread adds elements into the map, the other go along the given map and sum the values. Threads should work before catching ConcurrentModificationException. Try to fix the problem with ConcurrentHashMap and Collections.synchronizedMap(). What has happened after simple Map implementation exchanging? How it can be fixed in code? Try to write your custom ThreadSafeMap with synchronization and without. Run your samples with different versions of Java (6, 8, and 10, 11) and measure the performance. Provide a simple report to your mentor.

**Task 2 - Deadlocks**

**Cost**: 1 points.

Create three threads:

* 1st thread is infinitely writing random number to the collection;
* 2nd thread is printing sum of the numbers in the collection;
* 3rd is printing square root of sum of squares of all numbers in the collection.

Make these calculations thread-safe using synchronization block. Fix the possible deadlock.

**Task 3 - Where’s Your Bus, Dude?**

**Cost**: 1 points.

Implement message bus using Producer-Consumer pattern.

1. Implement asynchronous message bus. Do not use queue implementations from java.util.concurrent.
2. Implement producer, which will generate and post randomly messages to the queue.
3. Implement consumer, which will consume messages on specific topic and log to the console message payload.
4. (Optional) Application should create several consumers and producers that run in parallel.

**Task 4**

**Cost**: 1 points.

Create simple object pool with support for multithreaded environment. No any extra inheritance, polymorphism or generics needed here, just implementation of simple class:

/\*\*

\* Pool that block when it has not any items or it full

\*/

public class BlockingObjectPool {

/\*\*

\* Creates filled pool of passed size

\*

\* @param size of pool

\*/

public BlockingObjectPool(int size) {

...

}

/\*\*

\* Gets object from pool or blocks if pool is empty

\*

\* @return object from pool

\*/

public Object get() {

...

}

/\*\*

\* Puts object to pool or blocks if pool is full

\*

\* @param object to be taken back to pool

\*/

public void take(Object object) {

...

}

}

Use any blocking approach you like.

**Task 5**

**Cost**: 1 points.

Make an application that contains business logic for making exchange operations between different currencies.

1. Create models for dealing with currencies, user accounts and exchange rates. One account can have multiple currency values. Use BigDecimal for performing of exchange calculations.
2. Data with user accounts should be stored as files (one file per account).
3. Separate application functionality to DAO, service and utilities.
4. Create module which will provide high-level operations (manage accounts, currencies, exchange rates).
5. Create sample accounts and currencies. Define sample exchange rates.
6. Provide concurrent data access to user accounts. Simulate simultaneous currency exchanges for single account by multiple threads and ensure that all the operations are thread-safe.
7. Use ExecutorService to manage threads.
8. Make custom exceptions to let user to know the reason of error. Do not handle runtime exceptions.
9. Validate inputs such an account existence, sufficiency of currency amount, etc.
10. Log information about what is happening on different application levels and about conversion results. Use Logger for that.

**Task 6 (Optional)**

**Cost**: 1 points.

Create a multi-threading console application that starts two threads for producer and consumer respectively. It does not matter what kind of data it produces/consumes (e.g. producer could generate random numbers and consumer could calculate their total average). There must be a graceful shutdown (use Runtime.getRuntime().addShutdownHook(), Object's join()/interrupt() methods) to allow threads to correctly finish their work. When both producer and consumer are stopped print to console how many operations were performed per second (ops/sec).

This task should be implemented using two approaches:

1. Classic model: use non-blocking Queue implementation (e.g. LinkedList) to share data between producer and consumer threads use synchronized block, wait()/notify() methods to guard the queue from simultaneous access.
2. Concurrency use classes from java.util.concurrent package for synchronization (BlockingQueue, locks, etc.).

When both versions are done compare their performance (ops/sec).